**CSE-381: Operating Systems**

**Exercise #11**

Max Points: 20

**Note: If you are using your personal machine then prior to commencing work on this exercise, you may need to install XMing, Putty, and WinScp as illustrated in LinuxEnvironment.pdf (and shown in the videos in the Handouts folder).**

|  |
| --- |
| **Objective**: The objective of this exercise is to:   * Develop a data parallel application * Explore the effectiveness of using multiple threads for data parallel applications.   **Submission**: Save this MS-Word document using the naming convention *MUid*\_Exercise11.docx prior to proceeding with this exercise. Upload the following at the end of the lab exercise:   1. This MS-Word document saved with the convention *MUid*\_Exercise11.docx. 2. Program developed in his exercise named with the convention MUid\_Ex11.cpp.   You may discuss the questions with your instructor. |

# Preliminaries

1. Log onto the Linux server for this course via the following steps (that were covered in the previous exercises and as illustrated in the [LinuxEnvironment.pdf](https://niihka.muohio.edu/access/content/group/b05b7d30-d8c6-4312-9559-4d980565cbaf/Handouts%20_%20Video%20Tutorials/LinuxEvironment.pdf)):
   1. Run the X-Server Xming.
   2. Use PuTTY to log into the Linux server cse381-f12.csi.muohio.edu.
   3. When you log onto the server, you will be presented with a shell (**$**) prompt. You need to perform various tasks by typing commands at the shell prompt and pressing the enter (↵) key.
   4. Start emacs and ensure you see the graphical screen for emacs.

# Part #1: Convert given program to a data parallel version [10 points]

*Estimated time to complete: 35 minutes*

**Background**: Data parallel applications are essentially synchronization free (aka embarrassingly parallel) applications that are easy to parallelize using multiple threads. In a data parallel application, each thread operates on an independent sub-set of data. However, some level of synchronization is needed in data parallel applications to generate final outputs.

**Exercise**: The objective of this exercise is to convert a given C++ program into a suitable data parallel version in the following manner:

1. Download the supplied Exercise11.cpp and save it as MUid\_Ex11.cpp (where MUid is your login ID) program to the Linux server and briefly review the operation of the program.
2. Modify the program to accept the following two command-line arguments:
   1. The first command-line argument (referred as numPairs in this description) is the count of pairs of numbers to be randomly generated by your program.
   2. The second command-line argument (called numThreads in this description) is the count of number of threads to be used to compute GCD of the numbers generated by your program.
3. If numPairs is not a multiple of numThreads then report an error message stating “Number of pairs to be generated must be a multiple of number of threads” and exit.
4. The starter code in Exercise11.cpp merely uses two constant values. You need to modify the program to randomly generate pairs of numbers (using the rand() function used in Slide #6 in Synchronization.pptx slide deck off Niihka) to create numPairs number of ThreeInt objects in the numList vector. In other words, the numList vector must have numPair entries in it (once you have populated it) and each ThreeInt object must be initialized with ThreeInt{rand(), rand(), 0}.
5. Create numThreads number of threads to compute GCD for a subset of the numbers (similar to the strategy used in Slide #6 in Synchronization.pptx slide deck off Niihka).
6. Wait for the threads to complete.
7. Print the GCD values computed for the randomly generated pairs of numbers. The output should be the same immaterial of the number of threads used as shown in the sample output below.

|  |  |
| --- | --- |
|  | Ensure you compile your program to use the pthread library via the following command line to g++ (GCC’s C++ compiler):  g++ -std=c++0x -g -Wall raodm\_Ex11.cpp -o raodm\_Ex11 -lpthread |

**Sample output**:

Run your program with the same value for numPairs (first command-line argument) but different number of threads (second command-line argument). The output from your program should be consistent immaterial of the number of threads used:

|  |  |  |
| --- | --- | --- |
| **One Thread** |  | **Two Threads** |
| $ ./raodm\_Ex11 12 1  gcd(1804289383, 846930886) = 1  gcd(1681692777, 1714636915) = 1  gcd(1957747793, 424238335) = 1  gcd(719885386, 1649760492) = 2  gcd(596516649, 1189641421) = 1  gcd(1025202362, 1350490027) = 1  gcd(783368690, 1102520059) = 1  gcd(2044897763, 1967513926) = 1  gcd(1365180540, 1540383426) = 6  gcd(304089172, 1303455736) = 4  gcd(35005211, 521595368) = 1  gcd(294702567, 1726956429) = 3 |  | $ ./raodm\_Ex11 12 2  gcd(1804289383, 846930886) = 1  gcd(1681692777, 1714636915) = 1  gcd(1957747793, 424238335) = 1  gcd(719885386, 1649760492) = 2  gcd(596516649, 1189641421) = 1  gcd(1025202362, 1350490027) = 1  gcd(783368690, 1102520059) = 1  gcd(2044897763, 1967513926) = 1  gcd(1365180540, 1540383426) = 6  gcd(304089172, 1303455736) = 4  gcd(35005211, 521595368) = 1  gcd(294702567, 1726956429) = 3 |
|  |  |  |
| **Three Threads** |  | **Four Threads** |
| $ ./raodm\_Ex11 12 3  gcd(1804289383, 846930886) = 1  gcd(1681692777, 1714636915) = 1  gcd(1957747793, 424238335) = 1  gcd(719885386, 1649760492) = 2  gcd(596516649, 1189641421) = 1  gcd(1025202362, 1350490027) = 1  gcd(783368690, 1102520059) = 1  gcd(2044897763, 1967513926) = 1  gcd(1365180540, 1540383426) = 6  gcd(304089172, 1303455736) = 4  gcd(35005211, 521595368) = 1  gcd(294702567, 1726956429) = 3 |  | $ ./raodm\_Ex11 12 4  gcd(1804289383, 846930886) = 1  gcd(1681692777, 1714636915) = 1  gcd(1957747793, 424238335) = 1  gcd(719885386, 1649760492) = 2  gcd(596516649, 1189641421) = 1  gcd(1025202362, 1350490027) = 1  gcd(783368690, 1102520059) = 1  gcd(2044897763, 1967513926) = 1  gcd(1365180540, 1540383426) = 6  gcd(304089172, 1303455736) = 4  gcd(35005211, 521595368) = 1  gcd(294702567, 1726956429) = 3 |

# Part #2: Threading Analysis [10 points]

*Estimated time to complete: 45 minutes*

**Background**: The application being converted to a data parallel implementation is a CPU-bound task – *i.e.*, most of the runtime is spent in performing computations on the CPU. CPU-bound tasks can gain considerable performance improvements when run on a multi-core or multi-CPU machine such as cse381-f12.csi.muohio.edu that is backed by 4 real cores. However, to realize performance benefits a certain amount of computation needs to be performed in order to offset the overheads of multi-threading. The overheads include creation of threads, any synchronization overheads, and waiting for threads to finish.

In data parallel applications the synchronization overheads are very minimal, if not totally absent. Consequently, at least in theory, linear performance improvements are to be expected – *i.e.*, if the program takes *t* seconds to run with 1 thread, then ideally it would take only *t*/*n* seconds to run when run using *n* threads. This part of the exercise explores the effectiveness of data parallel implementation by comparing the time taken to run the program with varying number of threads.

Linux provides a time utility to measuring the time taken to run a program that can be used as shown in the sample output below (the actual timings you observe will be different and that is to be expected):

|  |
| --- |
| $ /usr/bin/time ./raodm\_Ex11 1200000 1 > /dev/null  3.80user 0.02system 0:03.83elapsed 99%CPU (0avgtext+0avgdata 51984maxresident)k  0inputs+0outputs (0major+3305minor)pagefaults 0swaps |

Each time a program is run, the actual time taken to run the program will vary depending on the load and other activities occurring on the system. Consequently, on multi-user, multi-tasking systems timing measurements have to be repeated in order to ensure that consistent timings are obtained and the consistent timings are averaged to obtain a suitable runtime value.

**Exercise**: Using the time command (shown above) measure and record runtime (to determine GCD of millions of randomly generated pairs of numbers) of the data-parallel version of the program in the table below. Ensure you run the program several times in each configuration and record only more-or-less consistent timings in the table. **The %CPU used for multiple threads should be above 100%.**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Timing Observations to compute GCD for 1,200,000 randomly generated pairs of numbers** | | | | | | |
| #Threads | Observation #1 | | Observation #2 | | Observation #3 | |
| Elapsed Time (sec) | %CPU | Elapsed Time (sec) | %CPU | Elapsed Time (sec) | %CPU |
| 1 |  |  |  |  |  |  |
| 2 |  |  |  |  |  |  |
| 3 |  |  |  |  |  |  |
| 4 |  |  |  |  |  |  |

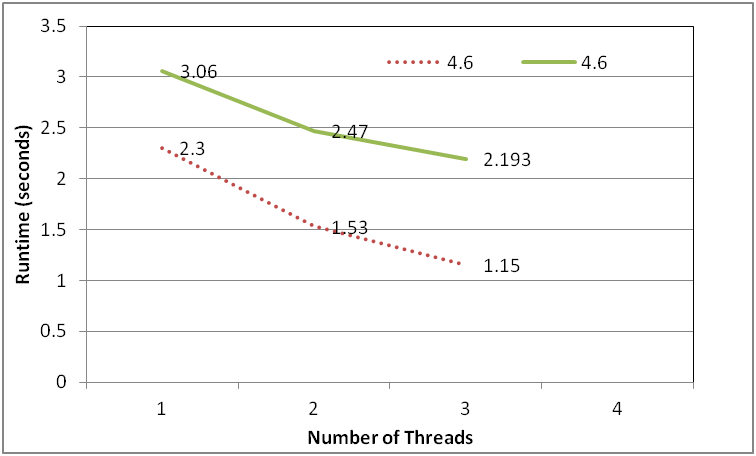
The average (of three consistent runs) runtime using one thread is assumed to the reference value. Compute the reference value by averaging the runtime for one thread recorded in the previous table.

|  |  |
| --- | --- |
| **The reference runtime value is:** |  |
| *(The reference runtime is the average runtime measured when using a single thread)* |  |

Using the timing data in the above table, compute and enter the average elapsed time values in the following table:

|  |  |  |
| --- | --- | --- |
| **Average (of 3 runs) Runtime (seconds)** | | |
| # Threads | Expected Theoretical Runtime  (seconds) | Observed Average Runtime  (seconds) |
| 1 | <*reference runtime*> | <*reference runtime*> |
| 2 | <*reference runtime*> / 2 | <average computed from previous table> |
| 3 | <*reference runtime*> / 3 | <average computed from previous table> |
| 4 | <*reference runtime*> / 4 | <average computed from previous table> |

Using the above data plot a chart (using Excel) with number of threads on X-axis and runtime on Y-axis similar to the example chart below. Copy-paste the chart you have generated replacing the chart shown below (the chart shown below is fictitious and your chart look different. **However, ensure you include axis titles, legend, and data-point labels in your chart!**).



Using the above chart answer the following question:

1. Are the observed average runtime and expected theoretical runtime the same? Do they follow a similar trend? What could plausibly explain any difference between the theoretical and observed behaviors of your program?

|  |
| --- |
|  |

# Part 3: Submit files to Niihka

Upload just the following files to Nihhka:

1. This MS-Word document saved with the convention *MUid*\_Exercise11.docx.
2. Program developed in Part #1 named with the convention MUid\_Ex11.cpp.